13 Programming for the NoSQL database service: DynamoDB

This chapter covers

- Advantages and disadvantages of the NoSQL service DynamoDB
- Creating tables and storing data
- Adding secondary indexes to optimize data retrieval
- Designing a data model optimized for a key-value database
- Tuning performance

Most applications depend on a database where data is stored. Imagine an application that keeps track of a warehouse's inventory. The more inventory moves through the warehouse, the more requests the application serves, and the more queries the database has to process. Sooner or later, the database becomes too busy and latency increases to a level that limits the warehouse's productivity. At this point, you have to scale the database to help the business. This can be done in two ways:

- *Vertically*—You can add more hardware to your database machine; for example, you can add memory or replace the CPU with a more powerful model.
- *Horizontally*—You can add a second database machine. Both machines then form a database cluster.

Scaling a database vertically is the easier option, but it gets expensive. High-end hardware is more expensive than commodity hardware. Besides that, at some point, you can no longer add faster hardware because nothing faster is available.

Scaling a traditional, relational database horizontally is difficult because transactional guarantees (atomicity, consistency, isolation, and durability, also known as *ACID*) require communication among all nodes of the database during a two-phase commit. A simplified two-phase commit with two nodes works like this:

1. A query is sent to the database cluster that wants to change data (INSERT, UPDATE, DELETE).
2. The database transaction coordinator sends a commit request to the two nodes.
3. Node 1 checks if the query could be executed. The decision is sent back to the coordinator. If the nodes decides yes, it must fulfill this promise. There is no way back.
4. Node 2 checks if the query could be executed. The decision is sent back to the coordinator.
5. The coordinator receives all decisions. If all nodes decide that the query could be executed, the coordinator instructs the nodes to finally commit.
6. Nodes 1 and 2 finally change the data. At this point, the nodes must fulfill the request. This step must not fail.

The problem is that the more nodes you add, the slower your database becomes, because more nodes must coordinate transactions between each other. The way to tackle this has been to use databases that don't adhere to these guarantees. They're called *NoSQL databases*.

There are four types of NoSQL databases—document, graph, columnar, and key-value store—each with its own uses and applications. Amazon provides a NoSQL database service called *DynamoDB*, a key-value store with document support. Unlike RDS, which effectively provides several common RDBMS engines like MySQL, MariaDB, Oracle Database, Microsoft SQL Server, and PostgreSQL, DynamoDB is a fully managed, proprietary, closed source key-value store with document support. *Fully managed* means that you only use the service and AWS operates it for you. DynamoDB is highly available and highly durable. You can scale from one item to billions and from one request per second to tens of thousands of requests per second.

If your data requires a different type of NoSQL database—a graph database like Neo4j, for example—you'll need to spin up an EC2 instance and install the database directly on that. Use the instructions in chapters 3 and 4 to do so.

This chapter looks in detail at how to use DynamoDB: both how to administer it like any other service, and how to program your applications to use it. Administering

DynamoDB is simple. You create tables and secondary indexes, and there is only one option to tweak: its read and write capacity, which directly affects its cost and its performance.

We'll look at the basics of DynamoDB and demonstrate them by walking through a simple to-do application called nodetodo, the Hello World of modern applications. Figure 13.1 shows nodetodo in action.

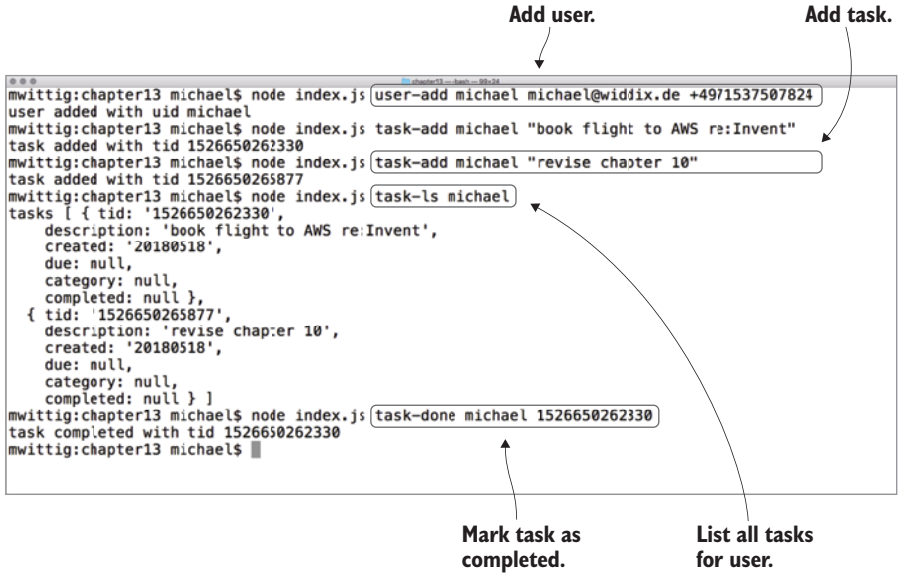


Figure 13.1 You can manage your tasks with the command-line to-do application nodetodo.

> **Examples are 100% covered by the Free Tier**
>
> The examples in this chapter are totally covered by the Free Tier. Keep in mind that this applies only if there is nothing else going on in your AWS account. You'll clean up your account at the end of the chapter.

Before you get started with nodetodo, you need to learn the basics of DynamoDB.

13.1 Operating DynamoDB

DynamoDB doesn't require administration like a traditional relational database, because it's a managed service and AWS takes care of that; instead, you have other tasks to take care of. Pricing depends mostly on your storage use and performance requirements. This section also compares DynamoDB to RDS.

13.1.1 Administration

With DynamoDB, you don't need to worry about installation, updates, machines, storage, or backups. Here's why:

- *DynamoDB isn't software you can download.* Instead, it's a NoSQL database as a service. Therefore you can't install DynamoDB like you would MySQL or MongoDB. This also means you don't have to update your database; the software is maintained by AWS.
- *DynamoDB runs on a fleet of machines operated by AWS.* AWS takes care of the OS and all security-related questions. From a security perspective, it's your job to grant the right permissions through IAM to the users of your DynamoDB tables.
- *DynamoDB replicates your data among multiple machines and across multiple data centers.* There is no need for a backup from a durability point of view—the backup is already built into the database.

> **Backups**
>
> DynamoDB provides very high durability. But what if the database administrator accidentally deletes all the data or a new version of the application corrupts items? In this case, you would need a backup to restore to a working table state from the past. In December 2017, AWS announced a new feature for DynamoDB: on-demand backup and restore.
>
> We strongly recommend using on-demand backups to create snapshots of your DynamoDB tables to be able to restore them later, if needed.

Now you know some administrative tasks that are no longer necessary if you use DynamoDB. But you still have things to consider when using DynamoDB in production: creating tables (see section 13.4), creating secondary indexes (section 13.6), monitoring capacity usage, provisioning read and write capacity (section 13.9), and creating backups of your tables.

13.1.2 Pricing

If you use DynamoDB, you pay the following monthly:

- $ 0.25 USD per used GB of storage (secondary indexes consume storage as well)
- $ 0.47 USD per provisioned write-capacity unit of throughput (throughput is explained in section 13.9)
- $ 0.09 USD per provisioned read-capacity unit of throughput

These prices were valid for the North Virginia (us-east-1) region, at the time of this writing. No additional traffic charges apply if you use AWS resources like EC2 instances to access DynamoDB in the same region.

13.1.3 Networking

DynamoDB does not run in your VPC. It is only accessible via the AWS API. You need internet access to talk to the AWS API. This means you can't access DynamoDB from a private subnet, because a private subnet has no route to the internet via an internet gateway. Instead, a NAT gateway is used (see section 6.5 for more details). Keep in mind that an application using DynamoDB can create a lot of traffic, and your NAT gateway is limited to 10 Gbps of bandwidth. A better approach is to set up a VPC endpoint for DynamoDB and use that to access DynamoDB from private subnets without needing a NAT gateway at all. You can read more about VPC endpoints in the AWS documentation at http://mng.bz/c4v6.

13.1.4 RDS comparison

Table 13.1 compares DynamoDB and RDS. Keep in mind that this is like comparing apples and oranges; the only thing DynamoDB and RDS have in common is that both are called databases. Use RDS (or to be more precise, the relational database engines offered by RDS) if your application requires complex SQL queries. Otherwise, you can consider migrating your application to DynamoDB.

Table 13.1 Differences between DynamoDB and RDS

Task	DynamoDB	RDS
Creating a table	Management Console, SDK, or CLI `aws dynamodb create-table`	SQL `CREATE TABLE` statement
Inserting, updating, or deleting data	SDK	SQL `INSERT, UPDATE, or DELETE` statement, respectively
Querying data	If you query the primary key: SDK. Querying non-key attributes isn't possible, but you can add a secondary index or scan the entire table.	SQL `SELECT` statement
Increasing storage	No action needed: DynamoDB grows with your items.	Provision more storage.
Increasing performance	Horizontal, by increasing capacity. DynamoDB will add more machines under the hood.	Vertical, by increasing instance size and disk throughput; or horizontal, by adding read replicas. There is an upper limit.
Installing the database on your machine	DynamoDB isn't available for download. You can only use it as a service.	Download MySQL, MariaDB, Oracle Database, Microsoft SQL Server, or PostgreSQL, and install it on your machine. Aurora is an exception.
Hiring an expert	Search for special DynamoDB skills.	Search for general SQL skills or special skills, depending on the database engine.

13.1.5 NoSQL comparison

Table 13.2 compares DynamoDB to several NoSQL databases. Keep in mind that all of these databases have pros and cons, and the table shows only a high-level comparison of how they can be used on top of AWS.

Table 13.2 Differences between DynamoDB and some NoSQL databases

Task	DynamoDB Key-value store	MongoDB Document store	Neo4j Graph store	Cassandra Columnar store	Riak KV Key-value store
Run the database on AWS in production.	One click: it's a managed service	Self-maintained cluster of EC2 instances, or as a service from a third party	Self-maintained cluster of EC2 instances, or as a service from a third party	Self-maintained cluster of EC2 instances, or as a service from a third party	Self-maintained cluster of EC2 instances, or as a service from a third party
Increase available storage while running.	Not necessary. The database grows automatically.	Add more EC2 instances.	Increase EBS volumes while running.	Add more EC2 instances.	Add more EC2 instances.

13.2 DynamoDB for developers

DynamoDB is a key-value store that organizes your data into tables. For example, you can have a table to store your users and another table to store tasks. The items contained in the table are identified by a primary key. An item could be a user or a task; think of an item as a row in a relational database. A table can also maintain secondary indexes for data lookup in addition to the primary key, which is also similar to relational databases. In this section, you'll look at these basic building blocks of DynamoDB, ending with a brief comparison of NoSQL databases.

13.2.1 Tables, items, and attributes

Each DynamoDB table has a name and organizes a collection of items. An *item* is a collection of attributes, and an *attribute* is a name-value pair. The attribute value can be scalar (number, string, binary, Boolean), multivalued (number set, string set, binary set), or a JSON document (object, array). Items in a table aren't required to have the same attributes; there is no enforced schema. Figure 13.2 demonstrates these terms.

You can create a table with the Management Console, CloudFormation, SDKs, or the CLI. The following example shows how you'd create a table with the CLI (don't try to run this command now—you'll create a table later in the chapter). The table is named app-entity and uses the `id` attribute as the primary key:

The primary key is a partition key using the id attribute.

Choose a name for your table, like app-entity.

The attribute named id is of type string.

```
$ aws dynamodb create-table --table-name app-entity \
➥ --attribute-definitions AttributeName=id,AttributeType=S \
➥ --key-schema AttributeName=id,KeyType=HASH \
➥ --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

You'll learn about this in section 13.9.

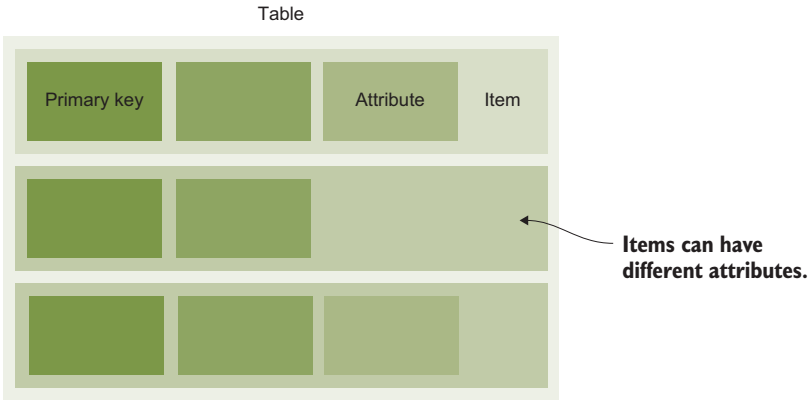


Figure 13.2 DynamoDB tables store items consisting of attributes identified by a primary key

If you plan to run multiple applications that use DynamoDB, it's good practice to prefix your tables with the name of your application. You can also add tables via the Management Console. *Keep in mind that you can't change the name of a table or its key schema later.* But you can add attribute definitions and change the throughput at any time.

13.2.2 Primary key

A primary key is unique within a table and identifies an item. You can use a single attribute as the primary key. DynamoDB calls this a *partition key*. You need an item's partition key to look up that item. You can also use two attributes as the primary key. In this case, one of the attributes is the partition key, and the other is called the *sort key*.

PARTITION KEY

A partition key uses a single attribute of an item to create a hash-based index. If you want to look up an item based on its partition key, you need to know the exact partition key. For example, a user table could use the user's email as a partition key. The user could then be retrieved if you know the partition key (email, in this case).

PARTITION KEY AND SORT KEY

When you use both a partition key and a sort key, you're using two attributes of an item to create a more powerful index. To look up an item, you need to know its exact partition key, but you don't need to know the sort key. You can even have multiple items with the same partition key: they will be sorted according to their sort key.

The partition key can only be queried using exact matches (=). The sort key can be queried using =, >, <, >=, <=, and BETWEEN x AND y operators. For example, you can query the sort key of a partition key from a certain starting point. You cannot query only the sort key—you must always specify the partition key. A message table could use a partition key and sort key as its primary key; the partition key could be the user's email, and the sort key could be a timestamp. You could then look up all of user's

messages that are newer or older than a specific timestamp, and the items would sorted according to the timestamp.

13.2.3 DynamoDB Local

Imagine a team of developers working on a new app using DynamoDB. During development, each developer needs an isolated database so as not to corrupt the other team members' data. They also want to write unit tests to make sure their app is working. You could create a unique set of DynamoDB tables with a CloudFormation stack for each developer. Or you could use a local DynamoDB for offline development. AWS provides an implementation of DynamoDB, which is available for download at http://mng.bz/27h5. Don't run it in production! It's only made for development purposes and provides the same functionality as DynamoDB, but it uses a different implementation: only the API is the same.

13.3 Programming a to-do application

To minimize the overhead of a programming language, you'll use Node.js/JavaScript to create a small to-do application that you can use via the terminal on your local machine. Let's call the application *nodetodo*. nodetodo will use DynamoDB as a database. With nodetodo, you can do the following:

- Create and delete users
- Create and delete tasks
- Mark tasks as done
- Get a list of all tasks with various filters

nodetodo supports multiple users and can track tasks with or without a due date. To help users deal with many tasks, tasks can be assigned to a category. nodetodo is accessed via the terminal. Here's how you would use nodetodo via the terminal to add a user. Important note: don't try to run the following commands now — they are not yet implemented. You will implement them in the following section.

Executes nodetodo in the terminal

Abstract description of the CLI: parameters are enclosed in < >.

```
# node index.js user-add <uid> <email> <phone>
$ node index.js user-add michael michael@widdix.de 0123456789
user added with uid michael
```

nodetodo's output is written to STDOUT.

To add a new task, you would do the following:

Named parameters are used with --name=value.

Optional parameters are enclosed in [ ].

```
# node index.js task-add <uid> <description> \
➥ [<category>] [--dueat=<yyyymmdd>]
$ node index.js task-add michael "plan lunch" --dueat=20150522
task added with tid 1432187491647
```

tid is the task ID.

You would mark a task as finished as follows:

```
# node index.js task-done <uid> <tid>
$ node index.js task-done michael 1432187491647
task completed with tid 1432187491647
```

You should also be able to list tasks. Here's how you would use nodetodo to do that:

```
# node index.js task-ls <uid> [<category>] [--overdue|--due|...]
$ node index.js task-ls michael
tasks [...]
```

To implement an intuitive CLI, nodetodo uses *docopt*, a command-line interface description language, to describe the CLI interface. The supported commands are as follows:

- `user-add`—Adds a new user to nodetodo
- `user-rm`—Removes a user
- `user-ls`—Lists users
- `user`—Shows the details of a single user
- `task-add`—Adds a new task to nodetodo
- `task-rm`—Removes a task
- `task-ls`—Lists user tasks with various filters
- `task-la`—Lists tasks by category with various filters
- `task-done`—Marks a task as finished

In the rest of the chapter, you'll implement those commands to learn about DynamoDB hands-on. This listing shows the full CLI description of all the commands, including parameters.

Listing 13.1 CLI description language docopt: using nodetodo (cli.txt)

```
nodetodo

Usage:
  nodetodo user-add <uid> <email> <phone>
  nodetodo user-rm <uid>
  nodetodo user-ls [--limit=<limit>] [--next=<id>]
  nodetodo user <uid>
  nodetodo task-add <uid> <description> \
➥ [<category>] [--dueat=<yyyymmdd>]
  nodetodo task-rm <uid> <tid>
  nodetodo task-ls <uid> [<category>] \
➥ [--overdue|--due|--withoutdue|--futuredue]
  nodetodo task-la <category> \
➥ [--overdue|--due|--withoutdue|--futuredue]
  nodetodo task-done <uid> <tid>
  nodetodo -h | --help
  nodetodo --version
```

The named parameters limit and next are optional.

The category parameter is optional.

Pipe indicates either/or.

help prints information about how to use nodetodo.

Version information

```
Options:
  -h --help         Show this screen.
  --version         Show version.
```

DynamoDB isn't comparable to a traditional relational database in which you create, read, update, or delete data with SQL. You'll access DynamoDB with an SDK to call the HTTPS REST API. You must integrate DynamoDB into your application; you can't take an existing application that uses an SQL database and run it on DynamoDB. To use DynamoDB, you need to write code!

13.4 Creating tables

Tables in DynamoDB organize your data. You aren't required to define all the attributes that the table items will have. DynamoDB doesn't need a static schema like a relational database does, but you must define the attributes that are used as the primary key in your table. In other words, you must define the table's primary key schema. To do so, you'll use the AWS CLI. The `aws dynamodb create-table` command has four mandatory options:

- `table-name`—Name of the table (can't be changed).
- `attribute-definitions`—Name and type of attributes used as the primary key. Multiple definitions can be given using the syntax `AttributeName=attr1, AttributeType=S`, separated by a space character. Valid types are `S` (String), `N` (Number), and `B` (Binary).
- `key-schema`—Name of attributes that are part of the primary key (can't be changed). Contains a single entry using the syntax `AttributeName=attr1,KeyType=HASH` for a partition key, or two entries separated by spaces for a partition key and sort key. Valid types are `HASH` and `RANGE`.
- `provisioned-throughput`—Performance settings for this table defined as `ReadCapacityUnits=5,WriteCapacityUnits=5` (you'll learn about this in section 13.9).

You'll now create a table for the users of the nodetodo application as well as a table that will contain all the tasks.

13.4.1 Users are identified by a partition key

Before you create a table for nodetodo users, you must think carefully about the table's name and primary key. We suggest that you prefix all your tables with the name of your application. In this case, the table name would be `todo-user`. To choose a primary key, you have to think about the queries you'll make in the future and whether there is something unique about your data items. Users will have a unique ID, called `uid`, so it makes sense to choose the `uid` attribute as the partition key. You must also be able to look up users based on the `uid` to implement the `user` command. Use a single attribute as primary key by marking the attribute as the partition key of your table. The following

example shows a user table where the attribute `uid` is used as the partition key of the primary key:

```
"michael" => {                    <-- uid ("michael") is the partition key; everything in { } is the item.
  "uid": "michael",
  "email": "michael@widdix.de",
  "phone": "0123456789"
}
"andreas" => {                    <-- Partition keys have no order.
  "uid": "andreas",
  "email": "andreas@widdix.de",
  "phone": "0123456789"
}
```

Because users will only be looked up based on the known `uid`, it's fine to use a partition key to identify a user. Next you'll create the user table, structured like the previous example, with the help of the AWS CLI:

```
$ aws dynamodb create-table --table-name todo-user \
➥ --attribute-definitions AttributeName=uid,AttributeType=S \
➥ --key-schema AttributeName=uid,KeyType=HASH \
➥ --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

Prefixing tables with the name of your application will prevent name clashes in the future.

Items must at least have one attribute uid of type string.

You'll learn about this in section 13.9.

The partition key (type HASH) uses the uid attribute.

Creating a table takes some time. Wait until the status changes to `ACTIVE`. You can check the status of a table as follows:

```
$ aws dynamodb describe-table --table-name todo-user
{
  "Table": {
    "AttributeDefinitions": [
      {
        "AttributeName": "uid",
        "AttributeType": "S"
      }
    ],
    "ProvisionedThroughput": {
      "NumberOfDecreasesToday": 0,
      "WriteCapacityUnits": 5,
      "ReadCapacityUnits": 5
    },
    "TableSizeBytes": 0,
    "TableName": "todo-user",
    "TableStatus": "ACTIVE",
    "KeySchema": [
      {
        "KeyType": "HASH",
        "AttributeName": "uid"
      }
```

CLI command to check the table status

Attributes defined for that table

Status of the table

Attributes used as the primary key

```
    ],
    "ItemCount": 0,
    "CreationDateTime": 1432146267.678
  }
}
```

13.4.2 Tasks are identified by a partition key and sort key

Tasks always belong to a user, and all commands that are related to tasks include the user's ID. To implement the `task-ls` command, you need a way to query the tasks based on the user's ID. In addition to the partition key, you can use the sort key. This way, you can add multiple items to the same partition key. Because all interactions with tasks require the user's ID, you can choose `uid` as the partition key and a task ID (`tid`), the timestamp of creation, as the sort key. Now you can make queries that include the user's ID and, if needed, the task's ID.

> **NOTE** This solution has one limitation: users can add only one task per timestamp. Because tasks are uniquely identified by `uid` and `tid` (the primary key) there can't be two tasks for the same user at the same time. Our timestamp comes with millisecond resolution, so it should be fine.

Using a partition key and a sort key uses two of your table attributes. For the partition key, an unordered hash index is maintained; the sort key is kept in a sorted index for each partition key. The combination of the partition key and the sort key uniquely identifies an item if they are used as the primary key. The following data set shows the combination of unsorted partition keys and sorted sort keys:

```
["michael", 1] => {
  "uid": "michael",
  "tid": 1,
  "description": "prepare lunch"
}
["michael", 2] => {
  "uid": "michael",
  "tid": 2,
  "description": "buy nice flowers for mum"
}
["michael", 3] => {
  "uid": "michael",
  "tid": 3,
  "description": "prepare talk for conference"
}
["andreas", 1] => {
  "uid": "andreas",
  "tid": 1,
  "description": "prepare customer presentation"
}
["andreas", 2] => {
  "uid": "andreas",
  "tid": 2,
  "description": "plan holidays"
}
```

uid ("michael") is the partition key and tid (1) is the sort key of the primary key.

The sort keys are sorted within a partition key.

There is no order in the partition keys.

nodetodo offers the ability to get all tasks for a user. If the tasks have only a partition key as the primary key, this will be difficult, because you need to know the partition key to extract them from DynamoDB. Luckily, using the partition key and sort key as the primary key makes things easier, because you only need to know the partition key to extract the items. For the tasks, you'll use `uid` as the known partition key. The sort key is `tid`. The task ID is defined as the timestamp when the task was created. You'll now create the task table, using two attributes to create a partition key and sort key as the primary key:

At least two attributes are needed for a partition key and sort key.

```
$ aws dynamodb create-table --table-name todo-task \
➥ --attribute-definitions AttributeName=uid,AttributeType=S \
➥ AttributeName=tid,AttributeType=N \
➥ --key-schema AttributeName=uid,KeyType=HASH \
➥ AttributeName=tid,KeyType=RANGE \
➥ --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

The tid attribute is the sort key.

Wait until the table status changes to `ACTIVE` when you run `aws dynamodb describe-table --table-name todo-task`. When both tables are ready, you'll add data.

13.5 Adding data

You have two tables up and running to store users and their tasks. To use them, you need to add data. You'll access DynamoDB via the Node.js SDK, so it's time to set up the SDK and some boilerplate code before you implement adding users and tasks.

Installing and getting started with Node.js

Node.js is a platform for executing JavaScript in an event-driven environment so you can easily build network applications. To install Node.js, visit https://nodejs.org and download the package that fits your OS. All examples in this book are tested with Node.js 8.

After Node.js is installed, you can verify if everything works by typing `node --version` into your terminal. Your terminal should respond with something similar to `v8.*`. Now you're ready to run JavaScript examples like nodetodo for AWS.

Do you want to get started with Node.js? We recommend *Node.js in Action* (Second Edition) by Alex Young, et al. (Manning, 2017), or the video course *Node.js in Motion* by P.J. Evans, (Manning, 2018).

To get started with Node.js and docopt, you need some magic lines to load all the dependencies and do some configuration work. Listing 13.2 shows how this can be done.

Where is the code located?

As usual, you'll find the code in the book's code repository on GitHub: https://github.com/AWSinAction/code2. nodetodo is located in /chapter13/. Switch to that directory, and run `npm install` in your terminal to install all needed dependencies.

Docopt is responsible for reading all the arguments passed to the process. It returns a JavaScript object, where the arguments are mapped to the parameters in the CLI description.

Listing 13.2 nodetodo: using docopt in Node.js (index.js)

```
const fs = require('fs');
const docopt = require('docopt');
const moment = require('moment');
const AWS = require('aws-sdk');
const db = new AWS.DynamoDB({
  region: 'us-east-1'
});

const cli = fs.readFileSync('./cli.txt', {encoding: 'utf8'});
const input = docopt.docopt(cli, {
  version: '1.0',
  argv: process.argv.splice(2)
});
```


Next you'll implement the features of nodetodo. You can use the `putItem` SDK operation to add data to DynamoDB like this:

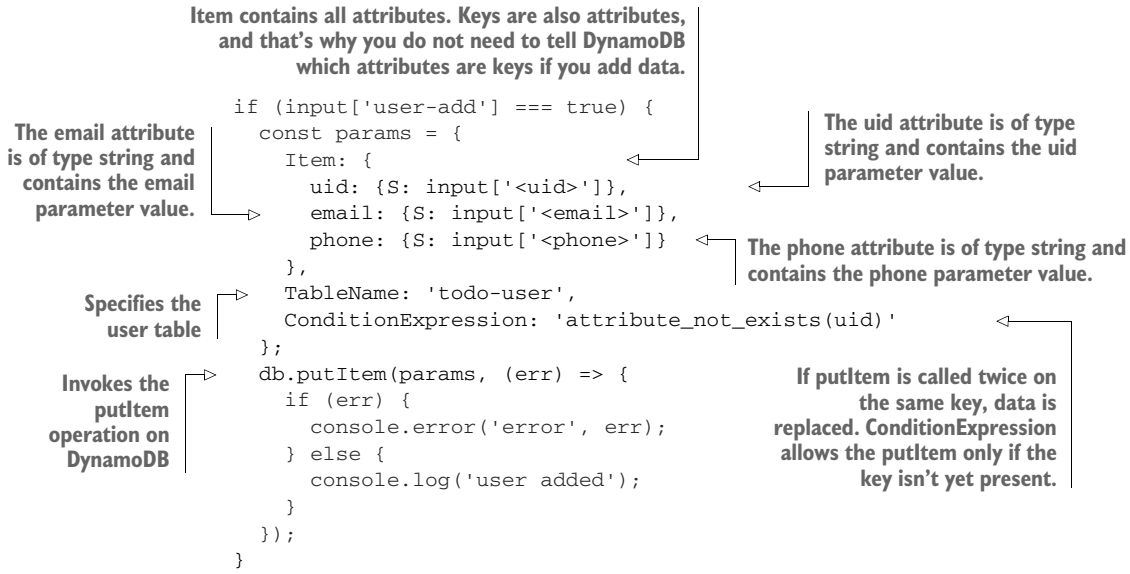
```
const params = {
  Item: {
    attr1: {S: 'val1'},
    attr2: {N: '2'}
  },
  TableName: 'app-entity'
};
db.putItem(params, (err) => {
  if (err) {
    console.error('error', err);
  } else {
    console.log('success');
  }
});
```


The first step is to add data to nodetodo.

13.5.1 Adding a user

You can add a user to nodetodo by calling `nodetodo user-add <uid> <email> <phone>`. In Node.js, you do this using this code.

Listing 13.3 nodetodo: adding a user (index.js)




```
if (input['user-add'] === true) {
  const params = {
    Item: {
      uid: {S: input['<uid>']},
      email: {S: input['<email>']},
      phone: {S: input['<phone>']}
    },
    TableName: 'todo-user',
    ConditionExpression: 'attribute_not_exists(uid)'
  };
  db.putItem(params, (err) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('user added');
    }
  });
}
```

When you make a call to the AWS API, you always do the following:

1. Create a JavaScript object (map) filled with the needed parameters (the params variable).
2. Invoke the function on the AWS SDK.
3. Check whether the response contains an error, and if not, process the returned data.

Therefore you only need to change the content of `params` if you want to add a task instead of a user.

13.5.2 Adding a task

You can add a task to nodetodo by calling `nodetodo task-add <uid> <description> [<category>] [--dueat=<yyyymmdd>]`. For example, to create a task to remember the milk, you could add a task like this: `nodetodo task-add michael "buy milk"`. In Node.js, you can implement this command with the code in listing 13.4.

Listing 13.4 nodetodo: adding a task (index.js)

```
if (input['task-add'] === true) {
  const tid = Date.now();
  const params = {
    Item: {
      uid: {S: input['<uid>']},
      tid: {N: tid.toString()},
      description: {S: input['<description>']},
      created: {N: moment(tid).format('YYYYMMDD')}
    },
    TableName: 'todo-task',
    ConditionExpression: 'attribute_not_exists(uid)
  and attribute_not_exists(tid)'
  };
  if (input['--dueat'] !== null) {
    params.Item.due = {N: input['--dueat']};
  }
  if (input['<category>'] !== null) {
    params.Item.category = {S: input['<category>']};
  }
  db.putItem(params, (err) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('task added with tid ' + tid);
    }
  });
}
```

Creates the task ID (tid) based on the current timestamp

The tid attribute is of type number and contains the tid value.

The created attribute is of type number (format 20150525).

Specifies the task table

If the optional named parameter dueat is set, add this value to the item.

Ensures that an existing item is not overridden

If the optional named parameter category is set, add this value to the item.

Invokes the putItem operation on DynamoDB

Now you can add users and tasks to nodetodo. Wouldn't it be nice if you could retrieve all this data?

13.6 Retrieving data

DynamoDB is a key-value store. The key is usually the only way to retrieve data from such a store. When designing a data model for DynamoDB, you must be aware of that limitation when you create tables (as you did in section 13.4). If you can use only one key to look up data, you'll sooner or later experience difficulties. Luckily, DynamoDB provides two other ways to look up items: a secondary index key lookup, and the scan operation. You'll start by retrieving data with its primary key and continue with more sophisticated methods of data retrieval.

DynamoDB Streams

DynamoDB lets you retrieve changes to a table as soon as they're made. A *stream* provides all write (create, update, delete) operations to your table items. The order is consistent within a partition key:

- If your application polls the database for changes, DynamoDB Streams solves the problem in a more elegant way.

- If you want to populate a cache with the changes made to a table, DynamoDB Streams can help.
- If you want to replicate a DynamoDB table to another region, DynamoDB Streams can do it.

13.6.1 Getting an item by key

The simplest form of data retrieval is looking up a single item by its primary key, for example a user by its ID. The `getItem` SDK operation to get a single item from DynamoDB can be used like this:

```
const params = {
  Key: {
    attr1: {S: 'val1'}
  },
  TableName: 'app-entity'
};
db.getItem(params, (err, data) => {
  if (err) {
    console.error('error', err);
  } else {
    if (data.Item) {
      console.log('item', data.Item);
    } else {
      console.error('no item found');
    }
  }
});
```

Specifies the attributes of the primary key

Invokes the getItem operation on DynamoDB

Checks whether an item was found

The command `nodetodo user <uid>` must retrieve a user by the user's ID (uid). Translated to the Node.js AWS SDK, this looks like the following listing.

Listing 13.5 nodetodo: retrieving a user (index.js)

```
const mapUserItem = (item) => {
  return {
    uid: item.uid.S,
    email: item.email.S,
    phone: item.phone.S
  };
};

if (input['user'] === true) {
  const params = {
    Key: {
      uid: {S: input['<uid>']}
    },
    TableName: 'todo-user'
  };
```

Helper function to transform DynamoDB result

Looks up a user by primary key

Specifies the user table

```
  db.getItem(params, (err, data) => {
    if (err) {
      console.error('error', err);
    } else {
      if (data.Item) {
        console.log('user', mapUserItem(data.Item));
      } else {
        console.error('user not found');
      }
    }
  });
}
```

Invokes the getItem operation on DynamoDB

Checks whether data was found for the primary key

You can also use the `getItem` operation to retrieve data by partition key and sort key, for example to look up a specific task. The only change is that the `Key` has two entries instead of one. `getItem` returns one item or no items; if you want to get multiple items, you need to query DynamoDB.

13.6.2 Querying items by key and filter

If you want to retrieve a collection of items rather than a single item, such as all tasks for a user, you must query DynamoDB. Retrieving multiple items by primary key only works if your table has a partition key and sort key. Otherwise, the partition key will only identify a single item. The `query` SDK operation to get a collection of items from DynamoDB can be used like this:

```
const params = {
  KeyConditionExpression: 'attr1 = :attr1val AND attr2 = :attr2val',
  ExpressionAttributeValues: {
    ':attr1val': {S: 'val1'},
    ':attr2val': {N: '2'}
  },
  TableName: 'app-entity'
};
db.query(params, (err, data) => {
  if (err) {
    console.error('error', err);
  } else {
    console.log('items', data.Items);
  }
});
```

Condition the key must match. Use AND if you're querying both a partition and sort key. Only the = operator is allowed for partition keys. Allowed operators for sort keys: =, >, <, >=, <=, BETWEEN x AND y, and begins_with. Sort key operators are blazing fast because the data is already sorted.

Dynamic values are referenced in the expression.

Always specify the correct type (S, N, B).

Invokes the query operation on DynamoDB

The `query` operation also lets you specify an optional `FilterExpression`, to include only items that match the filter and key condition. This is helpful to reduce the result set, for example to show only tasks of a specific category. The syntax of `FilterExpression` works like `KeyConditionExpression`, but no index is used for filters. Filters are applied to all matches that `KeyConditionExpression` returns.

To list all tasks for a certain user, you must query DynamoDB. The primary key of a task is the combination of the `uid` and the `tid`. To get all tasks for a user, `KeyConditionExpression` only requires the partition key. The implementation of `nodetodo task-ls <uid> [<category>] [--overdue|--due|--withoutdue|--futuredue]` is shown next.

Listing 13.6 nodetodo: retrieving tasks (index.js)

```
const getValue = (attribute, type) => {
  if (attribute === undefined) {
    return null;
  }
  return attribute[type];
};

const mapTaskItem = (item) => {
  return {
    tid: item.tid.N,
    description: item.description.S,
    created: item.created.N,
    due: getValue(item.due, 'N'),
    category: getValue(item.category, 'S'),
    completed: getValue(item.completed, 'N')
  };
};

if (input['task-ls'] === true) {
  const params = {
    KeyConditionExpression: 'uid = :uid',
    ExpressionAttributeValues: {
      ':uid': {S: input['<uid>']}
    },
    TableName: 'todo-task',
    Limit: input['--limit']
  };
  if (input['--next'] !== null) {
    params.KeyConditionExpression +=
      ' AND tid > :next';
    params.ExpressionAttributeValues[':next'] = {N: input['--next']};
  }
  if (input['--overdue'] === true) {
    params.FilterExpression = 'due < :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] = {N: yyyymmdd};
  } else if (input['--due'] === true) {
    params.FilterExpression = 'due = :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] = {N: yyyymmdd};
  } else if (input['--withoutdue'] === true) {
    params.FilterExpression = 'attribute_not_exists(due)';
  } else if (input['--futuredue'] === true) {
    params.FilterExpression = 'due > :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] = {N: yyyymmdd};
  } else if (input['--dueafter'] !== null) {
    params.FilterExpression = 'due > :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] =
```

Helper function to access optional attributes

Helper function to transform the DynamoDB result

Primary key query. The task table uses a partition and sort key. Only the partition key is defined in the query, so all tasks belonging to a user are returned.

Query attributes must be passed this way.

Filtering uses no index; it's applied over all elements returned from the primary key query.

Filter attributes must be passed this way.

attribute_not_exists(due) is true when the attribute is missing (opposite of attribute_exists).

```
      {N: input['--dueafter']};
  } else if (input['--duebefore'] !== null) {
    params.FilterExpression = 'due < :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] =
      {N: input['--duebefore']};
  }
  if (input['<category>'] !== null) {
    if (params.FilterExpression === undefined) {
      params.FilterExpression = '';
    } else {
      params.FilterExpression += ' AND ';
    }
    params.FilterExpression += 'category = :category';
    params.ExpressionAttributeValues[':category'] =
      S: input['<category>']};
  }
  db.query(params, (err, data) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('tasks', data.Items.map(mapTaskItem));
      if (data.LastEvaluatedKey !== undefined) {
        console.log('more tasks available with --next=' +
          data.LastEvaluatedKey.tid.N);
      }
    }
  });
}
```

Multiple filters can be combined with logical operators.

Invokes the query operation on DynamoDB

Two problems arise with the query approach:

1. Depending on the result size from the primary key query, filtering may be slow. Filters work without an index: every item must be inspected. Imagine you have stock prices in DynamoDB, with a partition key and sort key: the partition key is a ticker like AAPL, and the sort key is a timestamp. You can make a query to retrieve all stock prices of Apple (AAPL) between two timestamps (20100101 and 20150101). But if you only want to return prices on Mondays, you need to filter over all prices to return only 20% (1 out of 5 trading days each week) of them. That's wasting a lot of resources!
2. You can only query the primary key. Returning a list of all tasks that belong to a certain category for all users isn't possible, because you can't query the `category` attribute.

You can solve those problems with secondary indexes. Let's look at how they work.

13.6.3 Using global secondary indexes for more flexible queries

A *global secondary index* is a projection of your original table that is automatically maintained by DynamoDB. Items in an index don't have a primary key, just a key. This key is not necessarily unique within the index. Imagine a table of users where each user has a country attribute. You then create a global secondary index where the country is

the new partition key. As you can see, many users can live in the same country, so that key is not unique in the index.

You can query a global secondary index like you would query the table. You can imagine a global secondary index as a read-only DynamoDB table that is automatically maintained by DynamoDB: whenever you change the parent table, all indexes are asynchronously (eventually consistent!) updated as well. Figure 13.3 shows how a global secondary index works.

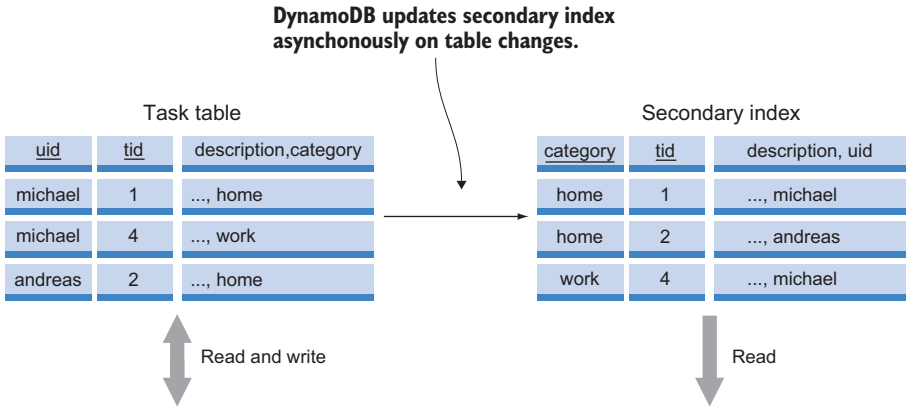


Figure 13.3 A global secondary index contains a copy (projection) of your table's data to provide fast lookup on another key.

A global secondary index comes at a price: the index requires storage (the same cost as for the original table). You must provision additional write-capacity units for the index as well, because a write to your table will cause a write to the global secondary index as well.

Local secondary index

Besides global secondary indexes, DynamoDB also support local secondary indexes. A local secondary index must use the same partition key as the table. You can only vary on the attribute that is used as the sort key. A local secondary index uses the read and write-capacity of the table.

A huge benefit of DynamoDB is that you can provision capacity based on your workload. If one of your global secondary indexes gets tons of read traffic, you can increase the read capacity of that index. You can fine-tune your database performance by provisioning sufficient capacity for your tables and indexes. You'll learn more about that in section 13.9.

Back to nodetodo. To implement the retrieval of tasks by category, you'll add a secondary index to the `todo-task` table. This will allow you to make queries by category.

A partition key and sort key are used: the partition key is the `category` attribute, and the sort key is the `tid` attribute. The index also needs a name: `category-index`. You can find the following CLI command in the README.md file in nodetodo's code folder:

```
$ aws dynamodb update-table --table-name todo-task \
➥ --attribute-definitions AttributeName=uid,AttributeType=S \
➥ AttributeName=tid,AttributeType=N \
➥ AttributeName=category,AttributeType=S \
➥ --global-secondary-index-updates '[{\
➥ "Create": {\
➥ "IndexName": "category-index", \
➥ "KeySchema": [{"AttributeName": "category", "KeyType": "HASH"}, \
➥ {"AttributeName": "tid", "KeyType": "RANGE"}], \
➥ "Projection": {"ProjectionType": "ALL"}, \
➥ "ProvisionedThroughput": {"ReadCapacityUnits": 5, \
➥ "WriteCapacityUnits": 5}\
➥ }}]'
```

You can add a global secondary index after the table is created.

Adds a category attribute, because the attribute will be used in the index.

Creates a new secondary index

The category attribute is the partition key, and the tid attribute is the sort key.

All attributes are projected into the index.

Creating a global secondary index takes some time. You can use the CLI to find out if the index is ready:

```
$ aws dynamodb describe-table --table-name=todo-task \
➥ --query "Table.GlobalSecondaryIndexes"
```

The following listing shows how the implementation of `nodetodo task-la <category> [--overdue|...]` uses the `query` operation.

Listing 13.7 nodetodo: retrieving tasks from a category index (index.js)

```
if (input['task-la'] === true) {
  const yyyymmdd = moment().format('YYYYMMDD');
  const params = {
    KeyConditionExpression: 'category = :category',
    ExpressionAttributeValues: {
      ':category': {S: input['<category>']}
    },
    TableName: 'todo-task',
    IndexName: 'category-index',
    Limit: input['--limit']
  };
  if (input['--next'] !== null) {
    params.KeyConditionExpression += ' AND tid > :next';
    params.ExpressionAttributeValues[':next'] = {N: input['--next']};
  }
  if (input['--overdue'] === true) {
    params.FilterExpression = 'due < :yyyymmdd';
    params.ExpressionAttributeValues[':yyyymmdd'] = {N: yyyymmdd};
  }
  [...]
  db.query(params, (err, data) => {
```

A query against an index works the same as a query against a table…

…but you must specify the index you want to use.

Filtering works the same as with tables

```
    if (err) {
      console.error('error', err);
    } else {
      console.log('tasks', data.Items.map(mapTaskItem));
      if (data.LastEvaluatedKey !== undefined) {
        console.log('more tasks available with --next='
          + data.LastEvaluatedKey.tid.N);
      }
    }
  });
}
```

But there are still situations where a query doesn't work: you can't retrieve all users. Let's look at what a table `scan` can do for you.

13.6.4 Scanning and filtering all of your table's data

Sometime you can't work with keys because you don't know them up front; instead, you need to go through all the items in the table. That's not very efficient, but in rare situations like daily batch jobs, it's okay. DynamoDB provides the `scan` operation to scan all items in a table:

```
const params = {
  TableName: 'app-entity',
  Limit: 50
};
db.scan(params, (err, data) => {
  if (err) {
    console.error('error', err);
  } else {
    console.log('items', data.Items);
    if (data.LastEvaluatedKey !== undefined) {
      console.log('more items available');
    }
  }
});
```

Specifies the maximum number of items to return

Invokes the scan operation on DynamoDB

Checks whether there are more items that can be scanned

The next listing shows the implementation of `nodetodo user-ls [--limit=<limit>] [--next=<id>]`. A paging mechanism is used to prevent too many items from being returned.

Listing 13.8 nodetodo: retrieving all users with paging (index.js)

```
if (input['user-ls'] === true) {
  const params = {
    TableName: 'todo-user',
    Limit: input['--limit']
  };
  if (input['--next'] !== null) {
    params.ExclusiveStartKey = {
      uid: {S: input['--next']}
    };
```

Maximum number of items returned

The named parameter next contains the last evaluated key.

```
  }
  db.scan(params, (err, data) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('users', data.Items.map(mapUserItem));
      if (data.LastEvaluatedKey !== undefined) {
        console.log('page with --next=' + data.LastEvaluatedKey.uid.S);
      }
    }
  });
}
```

Invokes the scan operation on DynamoDB

Checks whether the last item has been reached

The `scan` operation reads all items in the table. This example didn't filter any data, but you can use `FilterExpression` as well. Note that you shouldn't use the `scan` operation too often—it's flexible but not efficient.

13.6.5 Eventually consistent data retrieval

DynamoDB doesn't support transactions the same way a traditional database does. You can't modify (create, update, delete) multiple documents in a single transaction—the atomic unit in DynamoDB is a single item (to be more precise, a partition key).

In addition, DynamoDB is eventually consistent. That means it's possible that if you create an item (version 1), update that item to version 2, and then get that item, you may see the old version 1; if you wait and get the item again, you'll see version 2. Figure 13.4 shows this process. The reason for this behavior is because the item is persisted on multiple machines in the background. Depending on which machine answers your request, the machine may not have the latest version of the item.

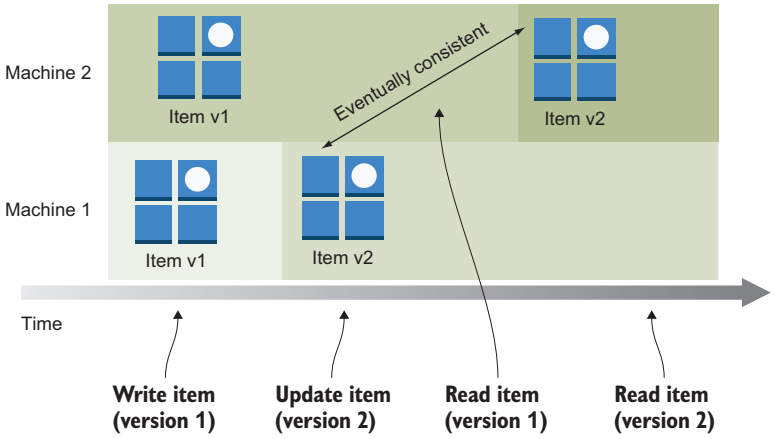


Figure 13.4 Eventually consistent reads can return old values after a write operation until the change is propagated to all machines.

You can prevent eventually consistent reads by adding `"ConsistentRead": true` to the DynamoDB request to get *strongly consistent reads*. Strongly consistent reads are supported by `getItem`, `query`, and `scan` operations. But a strongly consistent read takes longer and consumes more read capacity than an eventually consistent read. Reads from a global secondary index are always eventually consistent because the index itself is eventually consistent.

13.7 Removing data

Like the `getItem` operation, the `deleteItem` operation requires that you specify the primary key you want to delete. Depending on whether your table uses a partition key or a partition key and sort key, you must specify one or two attributes.

You can remove a user with nodetodo by calling `nodetodo user-rm <uid>`. In Node.js, you do this as shown in the following listing.

Listing 13.9 nodetodo: removing a user (index.js)

```
if (input['user-rm'] === true) {
  const params = {
    Key: {
      uid: {S: input['<uid>']}
    },
    TableName: 'todo-user'
  };
  db.deleteItem(params, (err) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('user removed');
    }
  });
}
```

Identifies an item by partition key

Specifies the user table

Invokes the deleteItem operation on DynamoDB

Removing a task is similar: `nodetodo task-rm <uid> <tid>`. The only change is that the item is identified by a partition key and sort key, and the table name has to be changed.

Listing 13.10 nodetodo: removing a task (index.js)

```
if (input['task-rm'] === true) {
  const params = {
    Key: {
      uid: {S: input['<uid>']},
      tid: {N: input['<tid>']}
    },
    TableName: 'todo-task'
  };
  db.deleteItem(params, (err) => {
    if (err) {
      console.error('error', err);
```

Identifies an item by partition key and sort key

Specifies the task table

```
    } else {
      console.log('task removed');
    }
  });
}
```

You're now able to create, read, and delete items in DynamoDB. The only operation you're missing is updating.

13.8 Modifying data

You can update an item with the `updateItem` operation. You must identify the item you want to update by its primary key; you can also provide an `UpdateExpression` to specify the updates you want to perform. You can use one or a combination of the following update actions:

- Use `SET` to override or create a new attribute. Examples: `SET attr1 = :attr1val`, `SET attr1 = attr2 + :attr2val`, `SET attr1 = :attr1val, attr2 = :attr2val`.
- Use `REMOVE` to remove an attribute. Examples: `REMOVE attr1`, `REMOVE attr1, attr2`.

In nodetodo, you can mark a task as done by calling `nodetodo task-done <uid> <tid>`. To implement this feature, you need to update the task item, as shown next in Node.js.

Listing 13.11 nodetodo: updating a task as done (index.js)

```
if (input['task-done'] === true) {
  const yyyymmdd = moment().format('YYYYMMDD');
  const params = {
    Key: {
      uid: {S: input['<uid>']},
      tid: {N: input['<tid>']}
    },
    UpdateExpression: 'SET completed = :yyyymmdd',
    ExpressionAttributeValues: {
      ':yyyymmdd': {N: yyyymmdd}
    },
    TableName: 'todo-task'
  };
  db.updateItem(params, (err) => {
    if (err) {
      console.error('error', err);
    } else {
      console.log('task completed');
    }
  });
}
```

Identifies the item by a partition and sort key

Defines which attributes should be updated

Attribute values must be passed this way.

Invokes the updateItem operation on DynamoDB

Cleaning up

Don't forget to delete your DynamoDB tables after you finish this section. Use the Management Console to do so.

That's it! You've implemented all of nodetodo's features.

13.9 Scaling capacity

When you create a DynamoDB table or a global secondary index, you must provision throughput. Throughput is divided into read and write capacity. DynamoDB uses `ReadCapacityUnits` and `WriteCapacityUnits` to specify the throughput of a table or global secondary index. But how is a capacity unit defined?

13.9.1 Capacity units

To understand capacity units, let's start by experimenting with the CLI:

```
$ aws dynamodb get-item --table-name todo-user \
➥ --key '{"uid": {"S": "michael"}}' \
➥ --return-consumed-capacity TOTAL \
➥ --query "ConsumedCapacity"
{
    "CapacityUnits": 0.5,
    "TableName": "todo-user"
}
```

Tells DynamoDB to return the used capacity units

getItem requires 0.5 capacity units.

```
$ aws dynamodb get-item --table-name todo-user \
➥ --key '{"uid": {"S": "michael"}}' \
➥ --consistent-read --return-consumed-capacity TOTAL \
➥ --query "ConsumedCapacity"
{
    "CapacityUnits": 1.0,
    "TableName": "todo-user"
}
```

A consistent read…

…needs twice as many capacity units.

More abstract rules for throughput consumption are as follows:

- An eventually consistent read takes half the capacity of a strongly consistent read.
- A strongly consistent `getItem` requires one read capacity unit if the item isn't larger than 4 KB. If the item is larger than 4 KB, you need additional read capacity units. You can calculate the required read capacity units using `roundUP(itemSize / 4)`.
- A strongly consistent `query` requires one read capacity unit per 4 KB of item size. This means if your query returns 10 items, and each item is 2 KB, the item size is 20 KB and you need 5 read units. This is in contrast to 10 `getItem` operations, for which you would need 10 read capacity units.

- A write operation needs one write capacity unit per 1 KB of item size. If your item is larger than 1 KB, you can calculate the required write capacity units using `roundUP(itemSize)`.

If capacity units aren't your favorite unit, you can use the AWS Simple Monthly Calculator at http://aws.amazon.com/calculator to calculate your capacity needs by providing details of your read and write workload.

The provision throughput of a table or a global secondary index is defined in seconds. If you provision five read capacity units per second with `ReadCapacityUnits=5`, you can make five strongly consistent `getItem` requests for that table if the item size isn't larger than 4 KB per second. If you make more requests than are provisioned, DynamoDB will first throttle your request. If you make many more requests than are provisioned, DynamoDB will reject your requests.

It's important to monitor how many read and write capacity units you require. Fortunately, DynamoDB sends some useful metrics to CloudWatch every minute. To see the metrics, open the AWS Management Console, navigate to the DynamoDB service, and select one of the tables ❶ and click the Metrics tab ❷. Figure 13.5 shows the CloudFormation metrics for the `todo-user` table.

Increasing the provisioned throughput is possible whenever you like, but you can only decrease the throughput of a table four to nine times a day (a day in UTC time). Therefore, you might need to overprovision the throughput of a table during some times of the day.

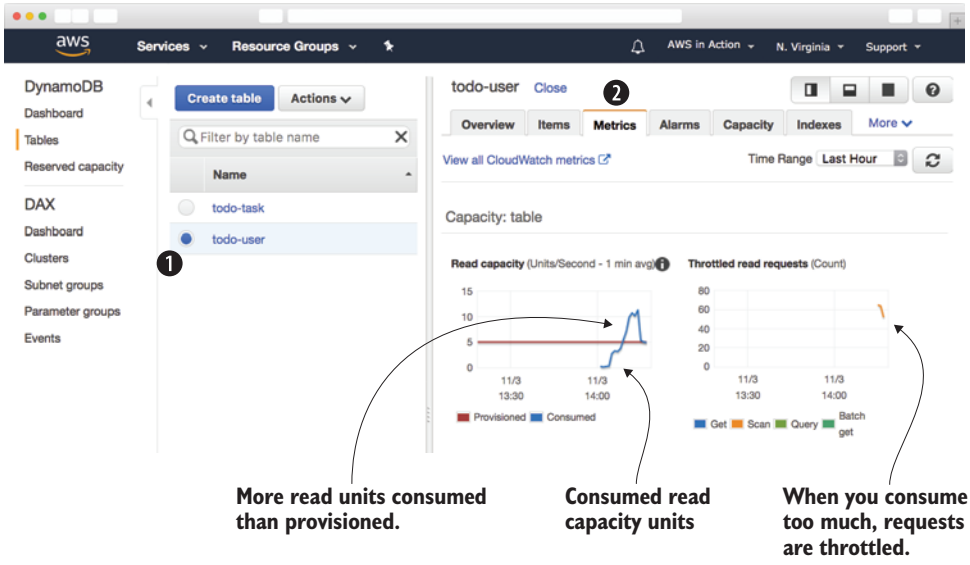


Figure 13.5 Monitoring provisioned and consumed capacity units of the DynamoDB table

Limits for decreasing the throughput capacity

Decreasing the throughput capacity of a table is generally only allowed four times a day (day in UTC time). Additionally, decreasing the throughout capacity is possible even if you have used up all four decreases in case the last decrease has happened more than four hours ago.

Theoretically, you can decrease the throughput capacity of your table up to nine times a day. When decreasing capacity four times in the first hour of the day, you get an additional decrease in the 5th hour. Next, after a decrease in the 5th hour, an additional decrease in the 11th hour is possible, and so on.

13.9.2 Auto-scaling

You can adjust the capacity of your DynamoDB tables and global secondary indexes based on your database's load. If your application tier scales automatically, it's a good idea to scale the database as well. Otherwise your database will become the bottleneck. The service used to implement this is called *Application Auto Scaling*. The following CloudFormation snippet shows how you can define auto-scaling rules:

```
# [...]
RoleScaling:
  Type: 'AWS::IAM::Role'
  Properties:
    AssumeRolePolicyDocument:
      Version: 2012-10-17
      Statement:
      - Effect: Allow
        Principal:
          Service: 'application-autoscaling.amazonaws.com'
        Action: 'sts:AssumeRole'
    Policies:
    - PolicyName: scaling
      PolicyDocument:
        Version: '2012-10-17'
        Statement:
        - Effect: Allow
          Action:
          - 'dynamodb:DescribeTable'
          - 'dynamodb:UpdateTable'
          - 'cloudwatch:PutMetricAlarm'
          - 'cloudwatch:DescribeAlarms'
          - 'cloudwatch:DeleteAlarms'
          Resource: '*'
TableWriteScalableTarget:
  Type: 'AWS::ApplicationAutoScaling::ScalableTarget'
  Properties:
    MaxCapacity: 20
    MinCapacity: 5
    ResourceId: 'table/todo-user'
    RoleARN: !GetAtt 'RoleScaling.Arn'
```

The IAM role is needed to allow AWS to adjust your tables.

Not more than 20 capacity units…

…not less than 5 capacity units…

…reference the DynamoDB table.

```
      ScalableDimension: 'dynamodb:table:WriteCapacityUnits'
      ServiceNamespace: dynamodb
  TableWriteScalingPolicy:
    Type: 'AWS::ApplicationAutoScaling::ScalingPolicy'
    Properties:
      PolicyName: TableWriteScalingPolicy
      PolicyType: TargetTrackingScaling
      ScalingTargetId: !Ref TableWriteScalableTarget
      TargetTrackingScalingPolicyConfiguration:
        TargetValue: 50.0
        ScaleInCooldown: 600
        ScaleOutCooldown: 60
        PredefinedMetricSpecification:
          PredefinedMetricType: DynamoDBWriteCapacityUtilization
```

Scale is write capacity units; you can also choose dynamodb:table:ReadCapacityUnits.

Wait at least 600 seconds between two scale ins (decrease in capacity).

Adjust capacity to reach an target utilization of 50%.

Wait at least 60 seconds between two scale outs (increase in capacity).

To help you explore DynamoDB auto-scaling, we created a CloudFormation template, located at http://mng.bz/3S89. Create a stack based on that template by clicking on the CloudFormation Quick-Create link at http://mng.bz/dsRI, and it will create the tables needed for the nodetodo application, including auto-scaling.

Figure 13.6 shows auto-scaling in action. At 18:10, the capacity is no longer sufficient. Therefore the capacity is automatically increased at 18:30.

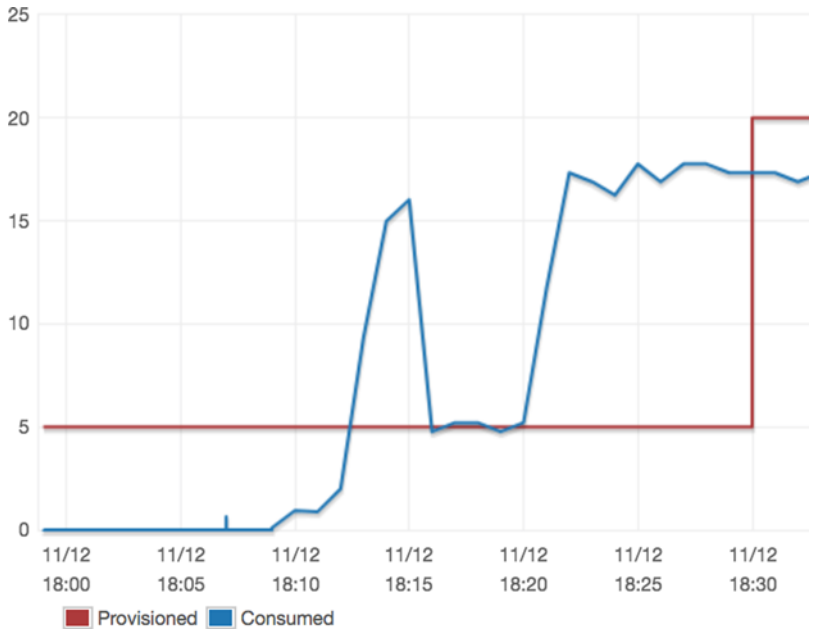


Figure 13.6 DynamoDB read capacity auto-scaling at work

Combine this knowledge of a database that scales with chapter 17, where you will learn to scale a fleet of EC2 instances. DynamoDB is the only database on AWS that grows and shrinks with your load.

Cleaning up

Don't forget to delete the CloudFormation stack `nodetodo`. Use the Management Console to do so.

Summary

- DynamoDB is a NoSQL database service that removes all the operational burdens from you, scales well, and can be used in many ways as the storage back end of your applications.
- Looking up data in DynamoDB is based on keys. You can only look up a partition key if you know the exact key. But DynamoDB also supports using a partition key and sort key, which combines the power of a partition key with another key that is sorted and supports range queries.
- You can enforce strongly consistent reads to avoid running into eventual consistency issues with stale data. But reads from a global secondary index are always eventually consistent.
- DynamoDB doesn't support SQL. Instead, you must use the SDK to communicate with DynamoDB from your application. This also implies that you can't use an existing application to run with DynamoDB without touching the code.
- Monitoring consumed read and write capacity is important if you want to provision enough capacity for your tables and indices.
- DynamoDB is charged per gigabyte of storage and per provisioned read or write capacity.
- You can use the `query` operation to query table or secondary indexes.
- The `scan` operation is flexible but not efficient and shouldn't be used too often.